CHAPTER

EIGHT

CONCURRENT PROCESSES

The work of George Milne and Robin Milner is an attempt to describe a mathematical semantics for concurrent computation and communication. Their goal is a formal calculus of concurrent computation, much as the lambda calculus is a formal calculus of uniprocess computation. They are trying to establish the meaning of programs, not to provide a tool for programming per se.

From the title of their paper, "Concurrent Processes and their Syntax," [Milne 79] we adopt the name "Concurrent Processes" for the Milne-Milner model. Their model has explicit processes that communicate synchronously and bidirectionally over labeled channels. The number of processes and their communication connections can change dynamically. Most of this work is concerned with proving that the semantics of their model is specific and unambiguous and that the communication operations provided by the model form an "algebra" with the right properties. This mathematics is heavily dependent on the theory of sets, powerdomains, and functions; it is beyond the scope of this book. Instead, we find their formalism interesting in its own right. We develop examples of that formalism to illustrate its power, reassured by the understanding that the resulting system is mathematically correct.*

* This chapter is perhaps the most difficult in the book. It is not essential for achieving an understanding of later material and can be skipped at first reading.

Processes and Nets

The fundamental processing objects of the Milne-Milner model are processes. Formally, a *process* is a set of ports. Each *port* is a triple, formed of a name, a value, and a continuation. We write the port with name α, value u, and continuation f as $\alpha{:}\langle \mathsf{u},\mathsf{f} \rangle$. Two ports are complementary if their names match, one "plain" and the other "barred." Thus, ports with names α and $\overline{\alpha}$ are complementary. Complementary ports can be "joined," providing a path for their processes to communicate.

Milne and Milner have a graphical notation to illustrate processes and communication paths. Figure 8-1 shows the graphic for process r. This process has two ports, α and $\overline{\beta}$. Port α has value u_1 and continuation f_1. Similarly, $\overline{\beta}$'s value is u_2 and its continuation, f_2.

When two processes communicate (compose), they simultaneously exchange information and reconfigure themselves into new processes. Processes communicate through complementary ports. If r communicates through the port α, it sends the value u_1 on the "communication line." It simultaneously receives a value on that line. Let us call that value v_1. The value v_1 becomes the (single) argument to the continuation f_1. The result of this application describes the new process (or set of processes). These continuations capture the state of the process, much as variables and the program counter capture the state of a conventional program. These notions of state and bidirectional communication are a formal echo of the ideas of Exchange Functions.

Milne and Milner use lambda expressions (Section 1-2) to describe the functions f_1 and f_2. They use the lambda calculus because it is a standard for mathematical description of functions, not because it has properties particularly amenable to programming. The important feature of the continuations f_1 and f_2 is that they evaluate to objects that describe processes (or sets of processes).

In Concurrent Processes, communicating processes send each other a value. This value can either include significant information or merely be a synchronization signal. Thus, there are four possible combinations of value transmission in communication: both processes send a signal (*pure synchronization*); a process receives a value, but sends a signal (*input*); a process sends a value, but receives a signal (*output*); or both processes send and receive values (*exchange*).

Concurrent Processes and Exchange Functions (Chapter 7) are the only systems we study that provide simultaneous, bidirectional communication. Most

Figure 8-1 A process.

Figure 8-2 A net of processes.

systems permit information transfer in only a single direction at a time. Even Milne and Milner note that the transmission of information (as opposed to signal) in both directions is not used by any of their examples (though it is used in one of ours). We surmise that bidirectional communication is included not so much as a reflection of "reality" but only to make the formalism more elegant.

A group of processes can be composed into a *net*. From the perspective of the external world, that net now acts (almost) like a single process. The compositions of the net then fall into three classes: *input*, in which a value is brought in from the outside environment; *output*, in which a value is sent to the environment; and *covert*, in which information passes between the processes internal to the net. Figure 8-2 shows the graphical description of a net of processes. The visible port names of a process characterize that process. This set is called the *sort* of the process. Similarly, when a group of processes is composed into a net, the port names externally visible from that net are the sort of the net. One operation in the model allows hiding the internal port names of the net.

In Concurrent Processes, the number of processes can grow without bound. The model allows a single composition to create (in a stepwise fashion) an unbounded number of new processes. Since this is a mathematical model, infinite sets are natural objects. The algebra of process composition determines which computations may take place. A "run" of a Concurrent Process system is thus the selection of one sequence of compositions from the many possible.

Figure 8-3 A pair of processes.

Mathematics of Composition

Let us assume that we have a pair of processes, p_1 and p_2, as diagrammed in Figure 8-3. Their composition, $p_1 | p_2$, is shown in Figure 8-4. Processes p_1 and p_2 connect through the complementary ports, γ and $\overline{\gamma}$. The composition $p_1|p_2$ has five communication capabilities with the external environment: α, $\overline{\alpha}$, $\overline{\beta}$, γ, and $\overline{\gamma}$. One of these is $\alpha{:}\langle u_1, \lambda w.(f_1(w))|p_2\rangle$. If this communication occurs, then u_1 is sent to the environment and a value x is sent to p_1. Process p_1 is transformed into $f_1(x)$. This is called a *renewal* of p_1. Process p_2 is unchanged by this communication; it is then composed with the renewal of p_1.

There is also the possibility of covert communication between p_1 and p_2 through their ports $\overline{\gamma}$ and γ. In that case, their composition also contains the composition of $f_2(v_3)$ and $g_3(u_2)$. This is the result of exchanging the values v_3 and u_2 and applying the continuations to the values received.

As precise as they may appear to be, the diagrams are only illustrative of the process composition relationship. The actual computation is specifically the result computed by the equations that define the model. The most important of these is the equation that defines process composition:*

Figure 8-4 The composition of p_1 and p_2.

* In these examples we use the set comprehension operator {}. However, the objects are powerdomains. To be technically correct we should use the powerdomain operators {‖}. A powerdomain is a power set with a partial-ordering relation among its members [Smyth 78].

$$\mathsf{p}|\mathsf{p}' = \quad \{\mu\colon \langle \mathsf{u}, \lambda \mathsf{v}.(\mathsf{f}(\mathsf{v})|\mathsf{p}')\rangle \text{ s.t. } \mu\colon \langle \mathsf{u}, \mathsf{f}\rangle \in \mathsf{p}\} \tag{1}$$

$$\cup \{\mu\colon \langle \mathsf{u}', \lambda \mathsf{v}'.(\mathsf{p}|\mathsf{f}'(\mathsf{v}'))\rangle \text{ s.t. } \mu\colon \langle \mathsf{u}', \mathsf{f}'\rangle \in \mathsf{p}'\} \tag{2}$$

$$\cup \bigcup \{\mathsf{f}(\mathsf{u}')|\mathsf{f}'(\mathsf{u}) \text{ s.t. } \mu\colon \langle \mathsf{u}, \mathsf{f}\rangle \in \mathsf{p}, \overline{\mu}\colon \langle \mathsf{u}', \mathsf{f}'\rangle \in \mathsf{p}'\} \tag{3}$$

The first two lines of this equation define a composition with the external environment. The first line, (1), specifies that process p is to be given a computing step and the result composed with p'. Line (2) is the corresponding stepping of p'. The final line, (3), describes the communication between p and p': the exchange of values and the recomposition of their respective continuations. This is a synchronous exchange.

Milne and Milner also provide mechanisms for renaming labels and for restricting label visibility, much as the lambda calculus provides α conversion for the renaming of lambda variables. Renaming is easier to understand by analogy to hardware design. Each process in the Milne-Milner model is like an integrated circuit and each label is like a pin on that circuit. One would no more want every α label on every process of type r to be connected than one would want every pin 3 on every shift register to share a common wire. Renaming and restricting visibility allow the correct connection of a process/label and its protection from similarly described processes. One wants to be able to bundle a net of processes together, secure that their internal connections cannot be "shorted" by an oddly named external wire. Like Lynch and Fischer, Milne and Milner prove that the behavior of a net of processes is indistinguishable from the behavior of a single process. Without relabeling and hiding this would not be possible.

The operator || first composes two nets (or processes) and then restricts their shared ports to be externally invisible.

Examples

In the following examples, we present the themes of Concurrent Processes without becoming mired in the mathematical details of formal semantics. Our programs may appear to be forbiddingly symbolic. However, they are usually just the lambda calculus expression of finite state automata or register automata.* In our most complicated example, the card reader, we provide state-transition diagrams of the modeled automata.

* Register automata are a variation of finite state automata in which registers that can store integers become part of the automata state and are used in determining state transitions. If the registers can store only a finite number of values, the resulting automata are still FSA; if they can store unbounded values (with the "right kind" of operations on those values), the automata are Turing-equivalent.

Figure 8-5 The initial state of the register.

Register A simple but interesting process in the Concurrent Processes model is a register. We consider a register r. This register responds to two different signals, set and get. Set takes the value sent and sets the register to that value, returning a synchronization signal. Get ignores its input, sends the register's value, and regenerates the register to the same register. "Setting the register to the value" is not exactly the correct description. Instead, sending a 5 to the register makes the register *become* the thing that responds to get with a 5, and to set x by *becoming* the thing that behaves like a register with value x. If the sequence of communications

$$\mathsf{set}\ 5,\ \mathsf{set}\ 12,\ \mathsf{get},\ \mathsf{get},\ \mathsf{set}\ 3,\ \mathsf{get}$$

is received, the register replies to the first two get's with 12, and the last get with 3. The register responds to the set commands by sending a synchronization pulse ($\odot$).

The register has ports set and $\overline{\mathsf{get}}$. Since it is meaningless to try to get the value of the register before it has been set, the register initially has only the port set. Figure 8-5 shows the initial state of the register. The register is

$$\mathsf{r} \equiv \{\mathsf{set}{:}\langle \odot,\ \lambda \mathsf{z}.\mathsf{REG}(\mathsf{z})\rangle\}$$

where the definition of REG is:*

$$\begin{aligned}\mathsf{REG}(\mathsf{z}) \equiv\ & \\ & \{\mathsf{set}{:}\langle \odot,\ \lambda \mathsf{z}'.\mathsf{REG}(\mathsf{z}')\rangle, \\ & \ \overline{\mathsf{get}}{:}\langle \mathsf{z},\ \mathsf{K}(\mathsf{REG}(\mathsf{z}))\rangle\}\end{aligned}$$

Evaluating REG(z) produces a process of two ports, set and $\overline{\mathsf{get}}$. Port set accepts a value z′ and regenerates the register; port $\overline{\mathsf{get}}$ responds with the value z (the value placed there by the last set) and regenerates the same register.

A typical sequence of communications with the register produces the register states shown in Table 8-1. To assign to the register, processes use ports labeled $\overline{\mathsf{set}}$; to retrieve the current value of the register, processes use ports labeled get. Processes that share this register are composed with it.

* K is the constant combinator, $\lambda \mathsf{x}.(\lambda \mathsf{y}.\mathsf{x})$. A constant function is a function that always returns the same value. For example, $f(x) = 5$ is a constant function. The constant combinator K constructs constant functions. What should the constant of these functions be? They take the value given as the argument to K. So K(5) is a function whose value is always 5, $\lambda \mathsf{y}.5$.

Table 8-1 Register states

Port	Value received			New register state	Value sent by register
Initial state			r ≡	{set: ⟨⊙, λz.REG(z)⟩}	
set	5	⇒	r ≡	{set: ⟨⊙, λz′.REG(z′)⟩, $\overline{\text{get}}$: ⟨5, λy.REG(5)⟩}	⊙
set	12	⇒	r ≡	{set: ⟨⊙, λz′.REG(z′)⟩, $\overline{\text{get}}$: ⟨12, λy.REG(12)⟩}	⊙
$\overline{\text{get}}$	⊙	⇒	r ≡	{set: ⟨⊙, λz′.REG(z′)⟩, $\overline{\text{get}}$: ⟨12, λy.REG(12)⟩}	12
$\overline{\text{get}}$	⊙	⇒	r ≡	{set: ⟨⊙, λz′.REG(z′)⟩, $\overline{\text{get}}$: ⟨12, λy.REG(12)⟩}	12
set	3	⇒	r ≡	{set: ⟨⊙, λz′.REG(z′)⟩, $\overline{\text{get}}$: ⟨3, λy.REG(3)⟩}	⊙

Binary semaphore In the previous example, the register started with a single port and quickly grew another. Our model of a binary semaphore is a process. It generates and eliminates a port with each step. Synchronization is assured because the semaphore has only a single legal port (P or V) at any time.

$$\text{free} \equiv \{\text{P}{:}\langle\odot, \text{K(busy)}\rangle\} \qquad \text{busy} \equiv \{\text{V}{:}\langle\odot, \text{K(free)}\rangle\}$$

When the semaphore is free it has a single port whose continuation takes it to the busy state; when it is busy it has a single port whose continuation takes it to the free state. Processes request the semaphore through their own port $\overline{\text{P}}$; they release the semaphore through port $\overline{\text{V}}$. The semaphore responds with a synchronization signal only, not information. Like the register, this semaphore uses the constant combinator K.

Theorem proving The Milne-Milner model allows the arbitrary growth (and shrinkage) of the process net. As an example of the creation of new processes in the Milne-Milner model, we examine a naive form of theorem proving over the propositional calculus.* This program is a partial decision procedure for tautologies in propositional logic. A *tautology* is a formula of propositional logic

* We direct the reader interested in the technology of automated theorem proving to Robinson [Robinson 79] or Loveland [Loveland 78].

(a *well-formed formula* or *wff*) that is true no matter what assignments are made to its variables. Thus, the formula

$$(P \wedge Q) \vee \neg(P \wedge Q)$$

is a tautology.

The algorithm requires first transforming the wff into conjunctive normal form. A formula in *conjunctive normal form* is the "**AND**ing" ($\wedge$) of a group of formulas, each of which is the "**OR**ing" ($\vee$) of a set of literals. A *literal* is either a propositional variable X or its negation $\neg X$. For example, the formula

$$(P \vee Q \vee \neg R) \wedge (\neg P \vee R) \wedge (Q)$$

is in conjunctive normal form. A theorem about propositional logic states that every formula is equivalent to some formula in conjunctive normal form. We call the literal $\neg X$ the *complement* of X, and X the complement of $\neg X$. A *clause* is a set of literals.

A pair of clauses, G and H, *clash* if there is some literal X such that X is in G and the complement of X is in H. The *fusion* of these two clauses is the clause

$$(G - \{X\}) \cup (H - \{\text{complement}(X)\})$$

For example, the fusion of the clauses:

$$\{A, \neg B, E\}$$

and

$$\{\neg C, D, \neg E\}$$

is the clause

$$\{A, \neg B, \neg C, D\}$$

A set of clauses *grows* if there are two clauses G and H in that set that clash. The set of clauses is then extended to include the result of the fusion.

The resolution algorithm takes a wff, negates it, and converts it into conjunctive normal form, revealing a clause structure. The algorithm then fuses pairs of clauses until it generates the *null clause* (the clause with no literals). The null clause indicates success; its appearance shows that the original formula is a tautology.

Real theorem proving systems take great care to avoid doing the same fusion repeatedly. However, we ignore this constraint in our simple program. Our procedure is also incomplete; it does not necessarily recognize that a particular formula is or is not a tautology. Instead, it continues processing, growing new processes at each communication exchange.

In our theorem prover we represent each clause as a process. Each literal in that clause is a port, with negative literals as the barred ports. At each port the

Figure 8-6 The resolution clause processes.

value offered is the clause, less the port literal. Each process regenerates both itself and a new process. This new process's program expresses the fusion of the two clauses. This process is added to the net and commences communicating with the other clauses. The net thereby grows. In general, the clause G=$\{e_1, e_2, \ldots e_n\}$ offers the communications

```
Clause(G) ≡
    (G = { }) →
        {answer:⟨Done, ⊙⟩},
        { e:⟨G−{e}, λG′.Clause (G′ ∪ (G−e))⟩ | Clause(G) s.t.  e ∈ G }
```

where $\neg e$ offers communication on $\overline{\mathsf{e}}$ (not shown in the code). Clause spawns two processes for the resolvent.* We present a simple example of the theorem prover. Consider the following well-formed formula of the propositional calculus:

$$((P \supset Q) \wedge (Q \supset R)) \supset (P \supset R)$$

We want to show that this formula is a tautology. We transform its negation into conjunctive normal form, yielding

$$(\neg P \vee Q) \wedge (\neg Q \vee R) \wedge (P) \wedge (\neg R)$$

which is rewritten in our set notation as

$$\{\neg P, Q\}\,\{\neg Q, R\}\,\{P\}\,\{\neg R\}.$$

The initial processes for the computation (illustrated in Figure 8-6) are

* The form $p \rightarrow x, y$ is an abbreviation for **if** p **then** x **else** y. This is a shorthand for conditionals (Section 1-2).

Figure 8-7 An intermediate resolution state.

$p_1 \equiv \{\overline{P}:\langle\{Q\}, \lambda u. \ldots \rangle$
$\quad Q:\langle\{\neg P\}, \lambda u. \ldots \rangle\}$

$p_2 \equiv \{\overline{Q}:\langle\{R\}, \lambda u. \ldots \rangle$
$\quad R:\langle\{\neg Q\}, \lambda u. \ldots \rangle\}$

$p_3 \equiv \{P:\langle\{\}, \lambda u. \ldots \rangle\}$

$p_4 \equiv \{\overline{R}:\langle\{\}, \lambda u. \ldots \rangle\}$

Each function regenerates its host process and a new process representing the fusion of its covert communication. If this clause is empty, instead of attempting to continue covert communication, that process has a port to relate the success of the process to the external environment. We leave completing the details of these functions as an exercise (Exercise 8-4).

Figure 8-7 shows a possible state of the net after several computational steps. Even if the original formula is a tautology, there is no guarantee that $p_1 | p_2 | p_3 | p_4$ ever communicates with the external environment.

Card reader Our final example of modeling with Concurrent Processes is the description of the interacting parts of a card reading system [Milne 78]. This example is particularly interesting because it models a system composed of both hardware and software components.

Table 8-2 The status register word

Bit	Status when set
15	Error.
14	Done reading. Another card may be demanded.
9	Card is being read.
8	Reader device off-line.
6	If set when status register loaded, allows the setting of bits 14 and 15 to cause a driver interrupt.
0	If set when status register loaded, causes driver to signal reader to begin reading.

The card reader is a Digital CR11. This reader is used in the PDP-11 series of computers. It follows the PDP-11 system philosophy of "peripheral device control through assignment to and interrogation of status registers." That is, to find out the state of a peripheral device (for example, waiting for a card, reading a card, or mutilating a card), the processor reads the status register associated with that device. Various bits of the register have different meanings. The processor forces the device to particular states by setting values in the status register.

The card reader has four components:

The *reader device* takes the card punches and translates them to numeric information usable by the rest of the system. We treat card reading as a primitive that reads an entire card in a single operation.

The *buffer register* receives the numeric values from the card reader, one at a time, and transmits them to the "outside world."

The *status register* is a 16-bit word. Each bit of the status register can represent some condition in the card reader hardware. Not all the bits in the CR11 status register are used. Table 8-2 lists the significant bits and their interpretation. In Figure 8-8, the significant bits are highlighted.

The *driver program* controls the sequencing of the other components. The driver program is software. This contrasts with the other components of the reader, which are all hardware.

Figure 8-9 shows the net of processes and connections for the card reader. In Figure 8-10, we redraw the connections between the card reader com-

Figure 8-8 The status bits.

Figure 8-9 The card reader net.

Figure 8-10 The flow of synchronization.

Figure 8-11 The reader device state machine.

ponents to show the flow of synchronization, control data, and information. The connecting links are labeled according to the variety of information transmitted: pure synchronization lines are dotted, control lines solid, and data lines double. The arrowheads show the direction of information flow. The whole net has three communication lines with the external environment: $\overline{\text{incard}}$, which connects to the card reading mechanism; out, which sends card values to the main computer; and enderr, which signals end-of-card and errors.

The reader device is a six-state automaton. When it receives a synchronization pulse (the 0-bit) from the status register (synch1), it responds by setting the 9-bit in the status register (up1, busy reading). It then inputs an entire card from the environment (incard) and waits for a next character pulse from the driver (synch2). It loops, sending characters to the buffer (val) at each next character pulse, until all 80 characters have been transmitted. It then sets the 14-bit in the status register (up1, ready for next card) and goes back to the synch1 state. Figure 8-11 shows the states of the reader device.

The program for the reader device is as follows:

```
reader-device ≡
    {synch1:⟨⊙,
          K(reader-device2)⟩}

reader-device2 ≡
    {up1:⟨⟨9,1⟩,
        K(reader-device3)⟩}

reader-device3 ≡
    {incard:⟨⊙,
          K(countsend(1))⟩}

countsend(j) ≡
    {synch2:⟨⊙,
          K((j=81) →
                {up1:⟨⟨14,1⟩,
                     K(reader-device)⟩},
                {val:⟨change(c[j]),
                     K(countsend(j+1))⟩})⟩}
```

where change is a function that encodes a card column as an integer and c is a card.

The buffer register is a three-state automaton. It simply loops between receiving a character from the reader device (val), getting a synchronization pulse from the driver (synch3), and sending the character to the outside environment (out). Figure 8-12 shows the state-transition diagram of the buffer register. The equations that define the buffer are

```
buffer-register ≡
    {val:⟨⊙,
        λn.{synch3:⟨⊙,
                K({out:⟨n,
                     K(buffer-register)⟩})⟩}⟩}
```

Figure 8-12 The buffer register state machine.

Figure 8-13 The status register state machine.

The status register is initially prepared for any of three different interactions. The driver can have an ask/ans dialogue with the status register, requesting the value of a particular bit. The driver can set (or clear) either the 0-bit or 6-bit in the status register, and the reader device can set either the 9-bit (indicating that it is busy reading a card) or the 14-bit (indicating that it has finished and is waiting). Figure 8-13 shows the states of the status register. The large left brace indicates the possible indeterminate interactions. The defining equations for the status register are as follows:

status-register $\equiv$
    status($\langle$0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0$\rangle$)

```
status(b) ≡
    {ask:⟨⊙,
        λn.{ans:⟨b[n+1],
                K(status(b))⟩}⟩,
     up2:⟨⊙,
        λm.((m[1]=0 ∧ m[2]=1) →
            {synch1:⟨⊙,
                K(status(⟨1, b[2], . . . , b[8], 0, . . . , 0⟩))⟩},
            status(⟨0, . . . , b[m[1]], m[2], b[m[1]+2], 0, . . . , 0⟩))⟩,
     up1:⟨⊙,
        λm.status(⟨0, . . . , b[m[1]], m[2], b[m[1]+2], . . . , b[16]⟩)⟩}
```

Here b is a 16-bit vector (the register) and m is an ordered pair. Vector b's elements are indexed from 1 to 16 (not 0 to 15). The first element of m (m[1]) is a bit number (0 . . . 15), and the second element (m[2]) is a set/clear (1/0 or true/false) value.

The final component of the card reader is the driver program. The program is a long loop. Most of the time it waits for an opportunity to read a card. When the card reader is no longer busy and is "online," the driver interrupt is enabled and the reading process is started. Reading continues in the embedded loop until the end-of-card message (or read error message) is received from the status register. The driver program then sends a -1 to the environment on the enderr line, disables the interrupt, and prepares to read the next card. Figure 8-14 shows this process. The defining equations for the driver program are as follows:

```
driver ≡
    {ask:⟨9,
        K({ans:⟨⊙,
            λt.((t=1) → driver, driver2)⟩})⟩}
driver2 ≡
    {ask:⟨8,
        K({ans:⟨⊙,
            λt.((t=1) → driver, driver3)⟩})⟩}
driver3 ≡
    {up2:⟨⟨6,1⟩,
        K({up2:⟨⟨0,1⟩,
            K(doio)⟩})⟩}
doio ≡
    {synch2:⟨⊙,
        K(doio2)⟩}
doio2 ≡
    {ask:⟨15,
        K(doio3)⟩}
```

Figure 8-14 The driver state machine.

doio3 ≡
    {$\overline{\text{ans}}$:⟨⊙,
        λt.((t=1) → endcard, doio4)⟩}
doio4 ≡
    {ask:⟨14,
        K(doio5)⟩}
doio5 ≡
    {$\overline{\text{ans}}$:⟨⊙,
        λt.((t=1) → endcard, doio6)⟩}
doio6 ≡
    {synch3:⟨⊙,
            K(doio)⟩}

```
endcard ≡
    {enderr:⟨-1,
            K(endcard2)⟩}
endcard2 ≡
    {up2:⟨⟨6,0⟩,
          K(driver)⟩}
```

The composition of these four devices forms the complete card reader device

$$\text{card reader} \equiv \text{reader-device} \parallel \text{status-register} \parallel \text{buffer-register} \parallel \text{driver}$$

where $\parallel$ is the "compose and restrict internal names" operator. It is possible to prove that the driver correctly controls the other components of the card reader (with respect to the appropriate specification).

Perspective

The Concurrent Processes model provides a formal semantics of concurrent computation. Stripped of its formalisms, this model specifies explicit processes that communicate bidirectionally and synchronously. Processes communicate over a set of ports (effectively, channels). Each port has two sides (barred and unbarred). Only processes seeking access from opposite sides of a port communicate. Processes in the Milne-Milner model can be dynamically created and destroyed. The model uses lambda expressions to describe the functional ability of the processes. Lambda expressions are used because they are the standard mathematical way of expressing functions; they have a long-studied and well-understood semantics.

PROBLEMS

8-1 Change the register to respond to set with the last value of the register.
8-2 Can you change the register to respond to set with the value sent?
8-3 Give an expression that represents a general (n-ary) semaphore.
8-4 Complete the code of Clause in the theorem prover.
8-5 Model an unbounded buffer in Concurrent Processes.
8-6 Write a version of the elevator controller of Chapter 14 using the communication mechanisms of Concurrent Processes. Which aspects of the elevator controller does the model capture? Which aspects is it unable to capture?

REFERENCES

[**Loveland 78**] Loveland, D. W., *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam (1978). This book is a detailed exposition on automated theorem proving, with particular emphasis on resolution and its variations.

[**Milne 78**] Milne, G. J., "A Mathematical Model of Concurrent Computation," Ph.D. dissertation, University of Edinburgh, Edinburgh (1978). This is Milne's doctoral dissertation, describing much of the model. Milne presented the original card reader example in this dissertation.

[**Milne 79**] Milne, G., and R. Milner, "Concurrent Processes and Their Syntax," *JACM*, vol. 26, no. 2 (April 1979), pp. 302–321. Milne and Milner present the definitive paper on Concurrent Processes. This paper is formal and difficult to read. The register and semaphore examples are drawn from this paper.

[**Milner 80**] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, New York (1980). CCS is another (and similar) attempt by Milner to develop an algebra of concurrent communicating systems.

[**Milner 83**] Milner, R., "Calculi for Synchrony and Asynchrony," *Theoret. Comp. Sci.*, vol. 25, no. 3 (1983), pp. 267–310. Milner presents a calculus for distributed computation based on four combinators. This paper extends his work on CCS [Milner 80] to include synchronous communication.

[**Robinson 79**] Robinson, J. A., *Logic: Form and Function*, North Holland, New York (1979). Like [Loveland 78], this is a book on automated theorem proving. Robinson places less emphasis on variations of resolution, and greater emphasis on the foundations of logic and semantics. He provides many programmed examples.

[**Smyth 78**] Smyth, M. B., "Powerdomains," *J. Comput. Syst. Sci.*, vol. 16, no. 1 (February 1978), pp. 23–36. Smyth describes the mathematics of powerdomains.